

O'REILLY®

Covers Swift 2
and Xcode 7



Free Sampler

iOS 9 Swift Programming Cookbook

SOLUTIONS & EXAMPLES FOR IOS APPS

Vandad Nahavandipoor

iOS 9 Swift Programming Cookbook

by Vandad Nahavandipoor

Copyright © 2016 Vandad Nahavandipoor. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Rachel Roumeliotis and Andy Oram

Production Editor: Nicole Shelby

Copyeditor: Kim Cofer

Proofreader: James Fraleigh

Indexer: Judy McConville

Interior Designer: David Futato

Cover Designer: Ellie Volckhausen

Illustrator: Rebecca Demarest

December 2015: First Edition

Revision History for the First Edition

2015-12-08: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491936696> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *iOS 9 Swift Programming Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-93669-6

[LSI]

Table of Contents

Preface	iii
1. Swift 2.0, Xcode 7, and Interface Builder	1
1.1 Handling Errors in Swift	1
1.2 Specifying Preconditions for Methods	3
1.3 Ensuring the Execution of Code Blocks Before Exiting Methods	4
1.4 Checking for API Availability	6
1.5 Categorizing and Downloading Assets to Get Smaller Binaries	7
1.6 Exporting Device-Specific Binaries	11
1.7 Linking Separate Storyboards Together	12
1.8 Adding Multiple Buttons to the Navigation Bar	13
1.9 Optimizing Your Swift Code	14
1.10 Showing the Header View of Your Swift Classes	18
1.11 Creating Your Own Set Types	19
1.12 Conditionally Extending a Type	20
1.13 Building Equality Functionality into Your Own Types	22
1.14 Looping Conditionally Through a Collection	24
1.15 Designing Interactive Interface Objects in Playgrounds	25
1.16 Grouping Switch Statement Cases Together	28
1.17 Bundling and Reading Data in Your Apps	28
2. Apple Watch	33
2.1 Downloading Files onto the Apple Watch	35
2.2 Noticing Changes in Pairing State Between the iOS and Watch Apps	39
2.3 Transferring Small Pieces of Data to and from the Watch	42
2.4 Transferring Dictionaries in Queues to and from the Watch	52
2.5 Transferring Files to and from the Watch	56
2.6 Communicating Interactively Between iOS and watchOS	60

2.7 Setting Up Apple Watch for Custom Complications	69
2.8 Constructing Small Complications with Text and Images	76
2.9 Displaying Time Offsets in Complications	87
2.10 Displaying Dates in Complications	94
2.11 Displaying Times in Complications	100
2.12 Displaying Time Intervals in Complications	106
2.13 Recording Audio in Your Watch App	112
2.14 Playing Local and Remote Audio and Video in Your Watch App	115
3. The User Interface.....	119
3.1 Arranging Your Components Horizontally or Vertically	119
3.2 Customizing Stack Views for Different Screen Sizes	121
3.3 Creating Anchored Constraints in Code	125
3.4 Allowing Users to Enter Text in Response to Local and Remote Notifications	130
3.5 Dealing with Stacked Views in Code	134
3.6 Showing Web Content in Safari View Controller	136
3.7 Laying Out Text-Based Content on Your Views	137
3.8 Improving Touch Rates for Smoother UI Interactions	138
3.9 Supporting Right-to-Left Languages	141
3.10 Associating Keyboard Shortcuts with View Controllers	146
3.11 Recording the Screen and Sharing the Video	147
4. Contacts.....	155
4.1 Creating Contacts	156
4.2 Searching for Contacts	161
4.3 Updating Contacts	166
4.4 Deleting Contacts	168
4.5 Formatting Contact Data	170
4.6 Picking Contacts with the Prebuilt System UI	174
4.7 Creating Contacts with a Prebuilt System UI	180
4.8 Displaying Contacts with a Prebuilt System UI	182
5. Extensions.....	185
5.1 Creating Safari Content Blockers	185
5.2 Creating Shared Links for Safari	190
5.3 Maintaining Your App's Indexed Content	193
6. Web and Search.....	199
6.1 Making Your App's Content Searchable	199
6.2 Making User Activities Searchable	203
6.3 Deleting Your App's Searchable Content	206

7. Multitasking	209
7.1 Adding Picture in Picture Playback Functionality	209
7.2 Handling Low Power Mode and Providing Alternatives	215
8. Maps and Location	219
8.1 Requesting the User’s Location a Single Time	219
8.2 Requesting the User’s Location in Background	221
8.3 Customizing the Tint Color of Pins on the Map	222
8.4 Providing Detailed Pin Information with Custom Views	225
8.5 Displaying Traffic, Scale, and Compass Indicators on the Map	227
8.6 Providing an ETA for Transit Transport Type	229
8.7 Launching the iOS Maps App in Transit Mode	232
8.8 Showing Maps in Flyover Mode	233
9. UI Testing	235
9.1 Preparing Your Project for UI Testing	235
9.2 Automating UI Test Scripts	238
9.3 Testing Text Fields, Buttons, and Labels	241
9.4 Finding UI Components	243
9.5 Long-Pressing on UI Elements	246
9.6 Typing Inside Text Fields	248
9.7 Swiping on UI Elements	250
9.8 Tapping UI Elements	251
10. Core Motion	253
10.1 Querying Pace and Cadence Information	254
10.2 Recording and Reading Accelerometer Data	255
11. Security	257
11.1 Protecting Your Network Connections with ATS	257
11.2 Binding Keychain Items to Passcode and Touch ID	259
11.3 Opening URLs Safely	261
11.4 Authenticating the User with Touch ID and Timeout	262
12. Multimedia	265
12.1 Reading Out Text with the Default Siri Alex Voice	265
12.2 Downloading and Preparing Remote Media for Playback	267
12.3 Enabling Spoken Audio Sessions	269
13. UI Dynamics	273
13.1 Adding a Radial Gravity Field to Your UI	273
13.2 Creating a Linear Gravity Field on Your UI	278

13.3 Creating Turbulence Effects with Animations	281
13.4 Adding Animated Noise Effects to Your UI	283
13.5 Creating a Magnetic Effect Between UI Components	285
13.6 Designing a Velocity Field on Your UI	288
13.7 Handling Nonrectangular Views	290
Index.....	297

Swift 2.0, Xcode 7, and Interface Builder

In this chapter, we are going to have a look at some of the updates to Swift (Swift 2.0), Xcode, and Interface Builder. We will start with Swift and some of the really exciting features that have been added to it since you read the *iOS 8 Swift Programming Cookbook*.

1.1 Handling Errors in Swift

Problem

You want to know how to throw and handle exceptions in Swift.



I'll be using *errors* and *exceptions* interchangeably in this book. When an error occurs in our app, we usually *catch* it, as you will soon see, and handle it in a way that is pleasant and understandable to the user.

Solution

To throw an exception, use the `throw` syntax. To catch exceptions, use the `do, try, catch` syntax.

Discussion

Let's say that you want to create a method that takes in a first name and last name as two arguments and returns a full name. The first name and the last name have to each at least be one character long for this method to work. If one or both have 0 lengths, we are going to want to throw an exception.

The first thing that we have to do is to define our errors of type `ErrorType`:

```
enum Errors : ErrorType{
    case EmptyFirstName
    case EmptyLastName
}
```

And then we are going to define our method to take in a first and last name and join them together with a space in between:

```
func fullNameFromFirstName(firstName: String,
    lastName: String) throws -> String{

    if firstName.characters.count == 0{
        throw Errors.EmptyFirstName
    }

    if lastName.characters.count == 0{
        throw Errors.EmptyLastName
    }

    return firstName + " " + lastName

}
```

The interesting part is really how to call this method. We use the `do` statement like so:

```
do{
    let fullName = try fullNameFromFirstName("Foo", lastName: "Bar")
    print(fullName)
} catch {
    print("An error occurred")
}
```

The `catch` clause of the `do` statement allows us to trap errors in a fine-grained manner. Let's say that you want to trap errors in the `Errors` enum differently from instances of `NSError`. Separate your `catch` clauses like this:

```
do{
    let fullName = try fullNameFromFirstName("Foo", lastName: "Bar")
    print(fullName)
}
catch let err as Errors{
    //handle this specific type of error here
    print(err)
}
catch let ex as NSError{
    //handle exceptions here
    print(ex)
}
catch {
    //otherwise, do this
}
```


See Also

Recipe 1.3

1.2 Specifying Preconditions for Methods

Problem

You want to make sure a set of conditions are met before continuing with the flow of your method.

Solution

Use the guard syntax.

Discussion

The guard syntax allows you to:

1. Specify a set of conditions for your methods.
2. Bind variables to optionals and use those variables in the rest of your method's body.

Let's have a look at a method that takes an optional piece of data as the `NSData` type and turns it into a `String` only if the string has some characters in it and is not empty:

```
func stringFromData(data: NSData?) -> String?{  
  
    guard let data = data,  
          let str = NSString(data: data, encoding: NSUTF8StringEncoding)  
          where data.length > 0 else{  
        return nil  
    }  
  
    return String(str)  
  
}
```

And then we are going to use it like so:

```
if let _ = stringFromData(nil){  
    print("Got the string")  
} else {  
    print("No string came back")  
}
```

We pass `nil` to this method for now and trigger the failure block (“No string came back”). What if we passed valid data? And to have more fun with this, let's create our

NSData instance this time with a guard. Because the NSString constructor we are about to use returns an optional value, we put a guard statement before it to ensure that the value that goes into the data variable is in fact a value, and not nil:

```
guard let data = NSString(string: "Foo")
    .dataUsingEncoding(NSUTF8StringEncoding) where data.length > 0 else{
    return
}

if let str = stringFromData(data){
    print("Got the string \(str)")
} else {
    print("No string came back")
}
```

So we can mix guard and where in the same statement. How about multiple let statements inside a guard? Can we do that? You betcha:

```
func example3(firstName: String?, lastName: String?, age: UInt8?){

    guard let firstName = firstName, let lastName = lastName, _ = age where
        firstName.characters.count > 0 && lastName.characters.count > 0 else{
        return
    }

    print(firstName, " ", lastName)

}
```

See Also

[Recipe 1.1](#)

1.3 Ensuring the Execution of Code Blocks Before Exiting Methods

Problem

You have various conditions in your method that can exit the method early. But before you do that, you want to ensure that some code always gets executed, for instance to do some cleanup.

Solution

Use the defer syntax.

Discussion

Anything that you put inside a `defer` block inside a method is guaranteed to get executed before your method returns to the caller. However, this block of code will get executed *after* the return call in your method. The code is also called when your method throws an exception.

Let's say that you want to define a method that takes in a string and renders it inside a new image context with a given size. Now if the string is empty, you want to throw an exception. However, before you do that, we want to make sure that we have ended our image context. Let's define our error first:

```
enum Errors : ErrorType{
    case EmptyString
}
```

Then we move onto our actual method that uses the `defer` syntax:

```
func imageForString(str: String, size: CGSize) throws -> UIImage{

    defer{
        UIGraphicsEndImageContext()
    }

    UIGraphicsBeginImageContextWithOptions(size, true, 0)

    if str.characters.count == 0{
        throw Errors.EmptyString
    }

    //draw the string here...

    return UIGraphicsGetImageFromCurrentImageContext()
}
```

I don't want to put `print()` statements everywhere in the code because it makes the code really ugly. So to see whether this really works, I suggest that you paste this code into your Xcode—or even better, grab the source code for this book's example code from GitHub, where I have already placed breakpoints in the `defer` and the `return` statements so that you can see that they are working properly.

You can of course then call this method like so:

```
do{
    let i = try imageForString("Foo", size: CGSize(width: 100, height: 50))
    print(i)
} catch let excep{
    print(excep)
}
```

See Also

Recipe 1.2

1.4 Checking for API Availability

Problem

You want to check whether a specific API is available on the host device running your code.

Solution

Use the `#available` syntax.

Discussion

We've all been waiting for this for a very long time. The days of having to call the `respondToSelector:` method are over (hopefully). Now we can just use the `#available` syntax to make sure a specific iOS version is available before making a call to a method.

Let's say that we want to write a method that can read an array of bytes from an `NSData` object. `NSData` offers a handy `getBytes:` method to do this, but Apple decided to deprecate it in iOS 8.1 and replace it with the better `getBytes:length:` version that minimizes the risk of buffer overflows. So assuming that one of our deployment targets is iOS 8 or older, we want to ensure that we call this new method if we are on iOS 8.1 or higher and the older method if we are on iOS 8.0 or older:

```
enum Errors : ErrorType{
    case EmptyData
}

func bytesFromData(data: NSData) throws -> [UInt8]{

    if (data.length == 0){
        throw Errors.EmptyData
    }

    var buffer = [UInt8](count: data.length, repeatedValue: 0)

    if #available(iOS 8.1, *){
        data.getBytes(&buffer, length: data.length)
    } else {
        data.getBytes(&buffer)
    }

    return buffer
}
```

```
}
```

And then we go ahead and call this method:

```
func example1(){  
  
    guard let data = "Foo".dataUsingEncoding(NSUTF8StringEncoding) else {  
        return  
    }  
  
    do{  
        let bytes = try bytesFromData(data)  
        print("Data = \(bytes)")  
    } catch {  
        print("Failed to get bytes")  
    }  
  
}
```

See Also

[Recipe 1.1](#)

1.5 Categorizing and Downloading Assets to Get Smaller Binaries

Problem

You have many assets in your app for various circumstances, and want to save storage space and network usage on each user’s device by shipping the app without the optional assets. Instead, you would want to dynamically download them and use them whenever needed.

Solution

Use Xcode to tag your assets and then use the `NSBundleResourceRequest` class to download them.

Discussion

For this recipe, I will create three packs of assets, each with three images in them. One pack may run for x3 screen scales, another for iPhone 6, and the last for iPhone 6+, for instance. I am taking very tiny clips of screenshots of my desktop to create these images—nothing special. The first pack will be called “level1,” the second “level2,” and the third “level3.”



Use the GitHub repo of this book for a quick download of the said resources. Also, for the sake of simplicity, I am assuming that we are going to run this only on x3 scale screens such as iPhone 6+.

Place all nine images (three packs of three images) inside your *Assets.xcassets* file and name them as shown in **Figure 1-1**. Then select all the images in your first asset pack and open the Attributes inspector. In the “On Demand Resource Tags” section of the inspector, enter **level1** and do the same thing for other levels—but of course bump the number up for each pack.

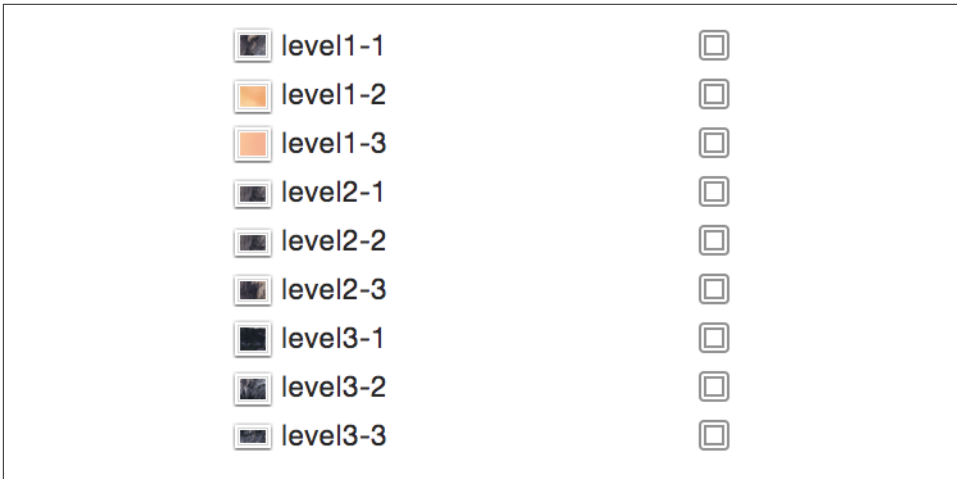


Figure 1-1. Name your assets as shown

Now, in your UI, place three buttons and three image views, hook the buttons’ actions to the code, and hook the image view references to the code:

```
@IBOutlet var img1: UIImageView!  
@IBOutlet var img2: UIImageView!  
@IBOutlet var img3: UIImageView!  
  
var imageViews: [UIImageView]{  
    return [self.img1, self.img2, self.img3]  
}
```

To find out whether the resource pack that you need has already been downloaded, call the `conditionallyBeginAccessingResourcesWithCompletionHandler` function on your resource request. Don’t blame me! I didn’t name this function. This will return a Boolean of `true` or `false` to tell you whether you have or don’t have access to the resource. If you don’t have access, you can simply download the resources with a

call to the `beginAccessingResourcesWithCompletionHandler` function. This will return an error if one happens, or `nil` if everything goes well.



We keep a reference to the request that we send for our asset pack so that the next time our buttons are tapped, we don't have to check their availability again, but release the previously downloaded resources using the `endAccessingResources` function.

```
var currentResourcePack: NSBundleResourceRequest?

func displayImagesForResourceTag(tag: String){
    NSOperationQueue.mainQueue().addOperationWithBlock{
        for n in 0..
```

```

    }
}

@IBAction func useLevel3(sender: AnyObject) {
    useLevel(3)
}

@IBAction func useLevel2(sender: AnyObject) {
    useLevel(2)
}

@IBAction func useLevel1(sender: AnyObject) {
    useLevel(1)
}

```

Run the code now in your simulator. When Xcode opens, go to the Debug Navigator (Command-6 key) and then click the Disk section. You will see something like that shown in [Figure 1-2](#).

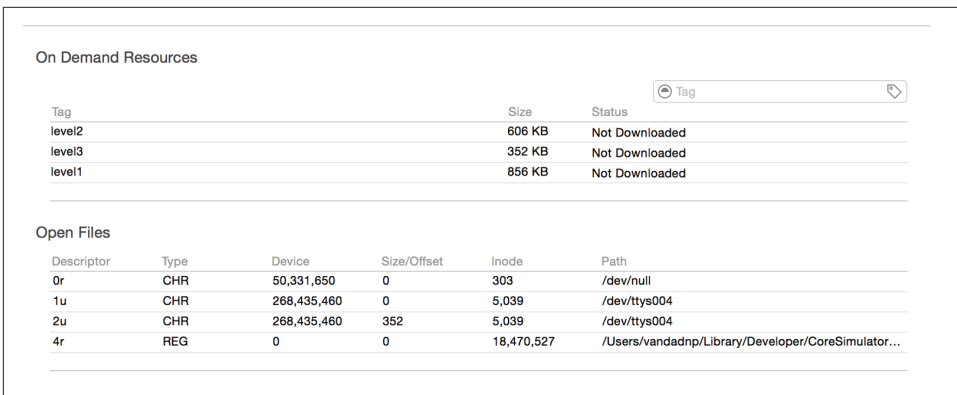


Figure 1-2. Xcode displaying all our On Demand Resources and status of whether or not they are downloaded locally

Note how none of the asset packs are in use. Now in your UI, click the first button to get the first asset pack and watch how the first asset pack's status will change to "In Use." Once you switch from that pack to another, the previously chosen pack will be set to "Downloaded" and be ready to be purged.

See Also

[Recipe 1.6](#)

1.6 Exporting Device-Specific Binaries

Problem

You want to extract your app's binary for a specific device architecture to find out how big your binary will be on that device when the user downloads your app.

Solution

Follow these steps:

1. Archive your app in Xcode.
2. In the Archives screen, click the Export button.
3. Choose the “Save for Ad Hoc Deployment” option in the new screen and click Next.
4. In the new window, choose “Export for specific device” and then choose your device from the list.
5. Once you are done, click the Next button and save your file to disk.

Discussion

With iOS 9, Apple introduced bitcode. This is Apple's way of specifying how the binary that you submit to the App Store will be downloaded on target devices. For instance, if you have an asset catalogue with some images for the iPad and iPhone and a second set of images for the iPhone 6 and 6+ specifically, users on iPhone 5 should not get the second set of assets. You don't have to do anything really to enable this functionality in Xcode 7. It is enabled by default. If you are working on an old project, you can enable bitcode from Build Settings in Xcode.

If you are writing an app that has a lot of images and assets for various devices, I suggest that you use this method, before submitting your app to the store, to ensure that the required images and assets are indeed included in your final build. Remember, if bitcode is enabled in your project, Apple will detect the host device that is downloading your app from the store and will serve the right binary to that device. You don't have to separate your binaries when submitting to Apple. You submit a big fat juicy binary and Apple will take care of the rest.

See Also

[Recipe 1.5](#)

1.7 Linking Separate Storyboards Together

Problem

You have a messy storyboard, so you would like to place some view controllers in their own storyboard and still be able to cross-reference them in your other storyboards.

Solution

Use IB's new "Refactor to Storyboard" feature under the Editor menu.

Discussion

I remember working on a project where we had a really messy storyboard and we had to separate the view controllers. What we ended up doing was putting the controllers on separate storyboards manually, after which we had to write code to link our buttons and other actions to the view controllers, instantiate them manually, and then show them. Well, none of that anymore. Apple has taken care of that for us!

As an exercise, create a single-view controller project in Xcode and then open your main storyboard. Then choose the Editor menu, then Embed In, and then Navigation Controller. Now your view controller has a navigation controller. Place a button on your view controller and then place another view controller on your storyboard. Select the button on the first view controller, hold down the Control button on your keyboard, drag the line over to the second view controller, and then choose the Show option. This will ensure that when the user taps your button, the system will push the second view controller onto the screen, as [Figure 1-3](#) shows.

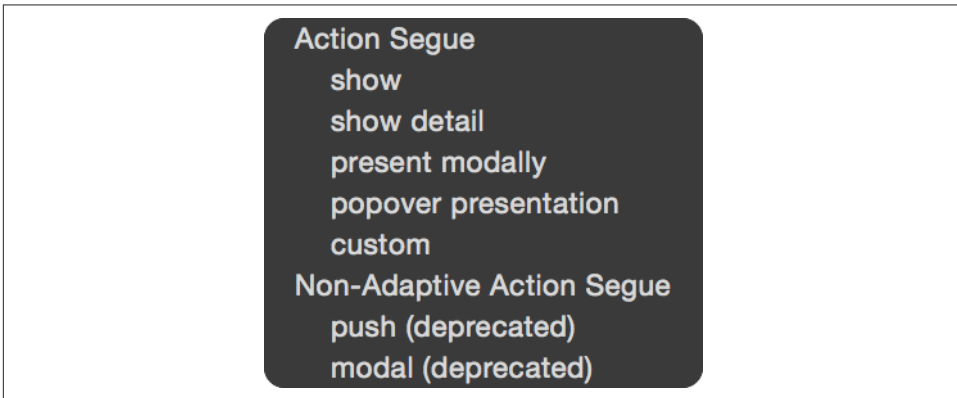


Figure 1-3. We need to create a show segue ensuring that pressing our button will show the second view controller

Now select your second view controller and then, from the Editor menu, choose the “Refactor to Storyboard” item. In the dialog, enter **Second.storyboard** as the file name and save. That’s really it. Now run your app and see the results if you want.

If you prefer to do some of this stuff manually instead of embedding things like this, you can always drag the new item called Storyboard Reference from the Object Library onto your storyboard and set up the name of the storyboard manually. Xcode will give you a drop-down box so that you don’t have to write the name of the storyboard all by yourself. You will also be able to specify an identifier for your storyboard. This identifier will then be useful when working with the segue. You of course have to set up this ID for your view controller in advance.

See Also

[Recipe 3.5](#)

1.8 Adding Multiple Buttons to the Navigation Bar

Problem

You want to add multiple instances of `UIBarButtonItem` to your navigation bar.

Solution

In Xcode 7, you can now add multiple bar button items to your navigation bar. Simply open the Object Library and search for “bar button.” Once you find the buttons, drag and drop them onto your navigation bar and then simply reference them in your code if you have to. For instance, [Figure 1-4](#) shows two bar buttons on the right-hand side of the navigation bar. In previous versions of Xcode, we could add only one button to each side. If we wanted more buttons, we had to write code to add them.

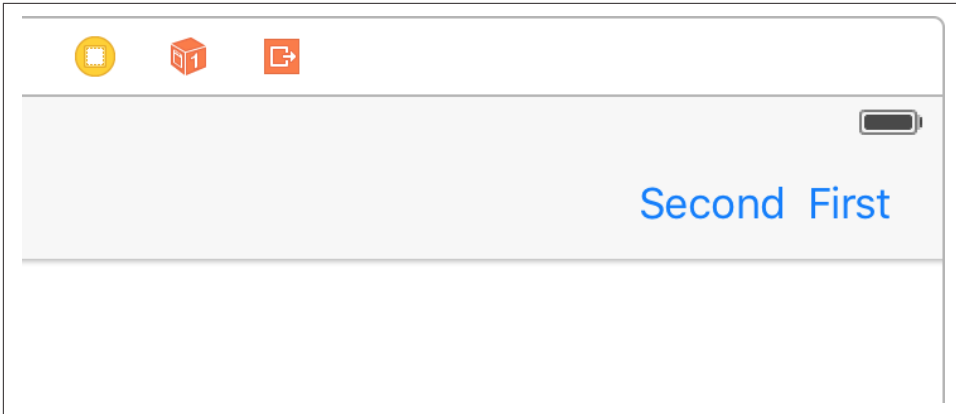


Figure 1-4. Two buttons on the same side of the navigation bar

Discussion

Prior to Xcode 7 you could not place multiple bar button items next to each other on your navigation bar. Well, now you can. You can also access these buttons just as you would expect, by creating a reference to them in your code. And you can always find them using the `barButtonItem`s property of your navigation bar.

See Also

[Recipe 1.7](#)

1.9 Optimizing Your Swift Code

Problem

You want to adopt some simple practices that can make your Swift code run much faster than before.

Solution

Use the following techniques:

1. Enable whole module optimization on your code.
2. Use value types (such as structs) instead of reference types where possible.
3. Consider using `final` for classes, methods, and variables that aren't going to be overridden.
4. Use the `CFAbsoluteTimeGetCurrent` function to profile your app inside your code.

5. Always use Instruments to profile your code and find bottlenecks.

Discussion

Let's have a look at an example. Let's say that we have a Person class like so:

```
class Person{
    let name: String
    let age: Int
    init(name: String, age: Int){
        self.name = name
        self.age = age
    }
}
```

Now we will write a method that will generate 100,000 instances of this class, place them inside a mutable array, and then enumerate the array. We will time this operation using the `CFAbsoluteTimeGetCurrent` function. We'll then be able to tell how many milliseconds this took:

```
func example1(){

    var x = CFAbsoluteTimeGetCurrent()

    var array = [Person]()

    for _ in 0..<100000{
        array.append(Person(name: "Foo", age: 30))
    }

    //go through the items as well
    for n in 0..
```

When I ran this code, it took 41.28 milliseconds to complete; it will probably be different in your computer. Now let's create a struct similar to the class we created before but without an initializer, because we get that for free. Then do the same that we did before and time it:

```
struct PersonStruct{
    let name: String
    let age: Int
}
```

```

func example2(){

    var x = CFAbsoluteTimeGetCurrent()

    var array = [PersonStruct]()

    for _ in 0..<100000{
        array.append(PersonStruct(name: "Foo", age: 30))
    }

    //go through the items as well
    for n in 0..

```



Don't suffix your struct names with "Struct" like I did. This is for demo purposes only, to differentiate between the class and the struct.

When I run this code, it takes only 35.53 milliseconds. A simple optimization brought some good savings. Also notice that in the release version these times will be massively improved, because your binary will have no debug information. I have tested the same code without the debugging, and the times are more around 4 milliseconds. Also note that I am testing these on the simulator, not on a real device. The profiling will definitely report different times on a device, but the ratio *should* be quite the same.

Another thing that you will want to do is think about which parts of your code are final and mark them with the `final` keyword. This will tell the compiler that you are not intending to override those properties, classes, or methods and will help Swift optimize the dispatch process. For instance, let's say we have this class hierarchy:

```

class Animal{
    func move(){
        if "Foo".characters.count > 0{
            //some code
        }
    }
}

class Dog : Animal{

```

```
}
```

And we create instances of the Dog class and then call the move function on them:

```
func example3(){
    var x = CFAbsoluteTimeGetCurrent()
    var array = [Dog]()
    for n in 0..<100000{
        array.append(Dog())
        array[n].move()
    }
    x = (CFAbsoluteTimeGetCurrent() - x) * 1000.0
    print("Took \(x) milliseconds")
}
```

When we run this, the runtime will first have to detect whether the move function is on the super class or the subclass and then call the appropriate class based on this decision. This checking takes time. For instance, if you know that the move function won't be overridden in the subclasses, mark it as final:

```
class AnimalOptimized{
    final func move(){
        if "Foo".characters.count > 0{
            //some code
        }
    }
}

class DogOptimized : AnimalOptimized{

}

func example4(){
    var x = CFAbsoluteTimeGetCurrent()
    var array = [DogOptimized]()
    for n in 0..<100000{
        array.append(DogOptimized())
        array[n].move()
    }
    x = (CFAbsoluteTimeGetCurrent() - x) * 1000.0
    print("Took \(x) milliseconds")
}
```

When I run these on the simulator, I get 90.26 milliseconds for the non-optimized version and 88.95 milliseconds for the optimized version. Not that bad.

I also recommend that you turn on whole module optimization for your release code. Just go to your Build Settings and under the optimization for your release builds (App Store scheme), simply choose “Fast” with Whole Module Optimization, and you are good to go.

See Also

[Recipe 1.1](#) and [Recipe 1.2](#)

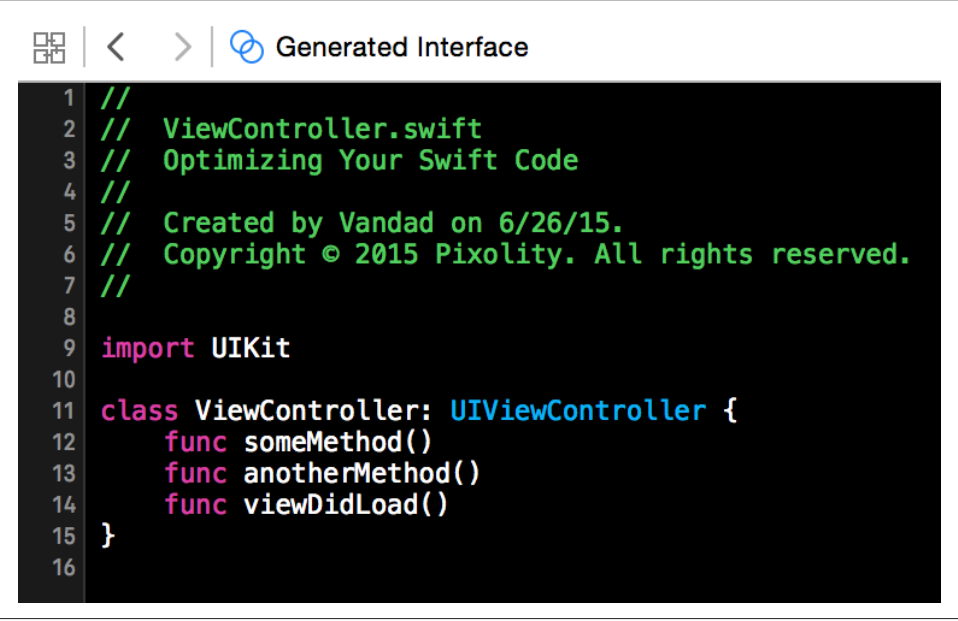
1.10 Showing the Header View of Your Swift Classes

Problem

You want to get an overview of what your Swift class's interface looks like.

Solution

Use Xcode's new Generated Interface Assistant Editor. This is how you do it. Open your Swift file first and then, in Xcode, use Show Assistant Editor, which you can find in the Help menu if you just type that name. After you open the assistant, you will get a split screen of your current view. Then in the second editor that opened, on top, instead of Counterparts (which is the default selection), choose Generated Interface. You'll see your code as shown in [Figure 1-5](#).



```
1 //
2 // ViewController.swift
3 // Optimizing Your Swift Code
4 //
5 // Created by Vandad on 6/26/15.
6 // Copyright © 2015 Pixolity. All rights reserved.
7 //
8
9 import UIKit
10
11 class ViewController: UIViewController {
12     func someMethod()
13     func anotherMethod()
14     func viewDidLoad()
15 }
16
```

Figure 1-5. Code shown in Xcode assistant

Discussion

I find the Generated Interface functionality of the assistant editor quite handy if you want to get an overview of how clean your code is. It probably won't be day-to-day

functionality that you use all the time, but I cannot be sure. Maybe you will love it so much that you will dedicate a whole new monitor just to see your generated interface all the time. By the way, there is a shortcut to the assistant editor in Xcode 7: Command-Alt-Enter. To get rid of the editor, press Command-Enter.

See Also

[Recipe 1.7](#)

1.11 Creating Your Own Set Types

Problem

You want to create a type in Swift that can allow all operators that normal sets allow, such as the `contains` function.

Solution

Conform to the `OptionSetType` protocol. As a bonus, you can also conform to the `CustomDebugStringConvertible` protocol, as I will do in this recipe, in order to set custom debug descriptions that the `print` function can use during debugging of your sets.

Discussion

Let's say that I have a structure that keeps track of iPhone models. I want to be able to create a set of this structure's values so that I can say that I have an iPhone 6, iPhone 6+, and iPhone 5s (fancy me!). Here is the way I would do that:

```
struct iPhoneModels : OptionSetType, CustomDebugStringConvertible{

    let rawValue: Int
    init(rawValue: Int){
        self.rawValue = rawValue
    }

    static let Six = iPhoneModels(rawValue: 0)
    static let SixPlus = iPhoneModels(rawValue: 1)
    static let Five = iPhoneModels(rawValue: 2)
    static let FiveS = iPhoneModels(rawValue: 3)

    var debugDescription: String{
        switch self{
        case iPhoneModels.Six:
            return "iPhone 6"
        case iPhoneModels.SixPlus:
            return "iPhone 6+"
        }
    }
}
```

```

        case IphoneModels.Five:
            return "iPhone 5"
        case IphoneModels.FiveS:
            return "iPhone 5s"
        default:
            return "Unknown iPhone"
    }
}
}

```

And then I can use it like so:

```

func example1(){

    let myIphones: [IphoneModels] = [.Six, .SixPlus]

    if myIphones.contains(.FiveS){
        print("You own an iPhone 5s")
    } else {
        print("You don't seem to have an iPhone 5s but you have these:")
        for i in myIphones{
            print(i)
        }
    }
}
}

```

Note how I could create a set of my new type and then use the `contains` function on it just as I would on a normal set. Use your imagination—this is some really cool stuff.

See Also

[Recipe 1.1](#), [Recipe 1.2](#), and [Recipe 1.3](#)

1.12 Conditionally Extending a Type

Problem

You want to be able to extend existing data types that pass a certain test.

Solution

Use protocol extensions. Swift 2.0 allows protocol extensions to contain code.

Discussion

Let's say that you want to add a method on any array in Swift where the items are integers. In your extension, you want to provide a method called `canFind` that can find a specific item in the array and return yes if it could be found. I know that we can do this with other system methods. I am offering this simple example to demonstrate how protocol extensions work:

```
extension SequenceType where
    Generator.Element : IntegerArithmeticType{
    public func canFind(value: Generator.Element) -> Bool{
        for (_, v) in self.enumerate(){
            if v == value{
                return true
            }
        }
        return false
    }
}
```

Then you can go ahead and use this method like so:

```
func example1(){

    if [1, 3, 5, 7].canFind(5){
        print("Found it")
    } else {
        print("Could not find it")
    }

}
```

As another example, let's imagine that you want to extend all array types in Swift (`SequenceType`) that have items that are either double or floating point. It doesn't matter which method you add to this extension. I am going to add an empty method for now:

```
extension SequenceType where Generator.Element : FloatingPointType{
    //write your code here
    func doSomething(){
        //TODO: code this
    }
}
```

And you can, of course, use it like so:

```
func example2(){

    [1.1, 2.2, 3.3].doSomething()

}
```

However, if you try to call this method on an array that contains non-floating-point data, you will get a compilation error.

Let me show you another example. Let's say that you want to extend all arrays that contain only strings, and you want to add a method to this array that can find the longest string. This is how you would do that:

```
extension SequenceType where Generator.Element : StringLiteralConvertible{
    func longestString() -> String{
        var s = ""
        for (_, v) in self.enumerate(){
            if let temp = v as? String
                where temp.characters.count > s.characters.count{
                s = temp
            }
        }
        return s
    }
}
```

Calling it is as simple as:

```
func example3(){
    print(["Foo", "Bar", "Vandad"].longestString())
}
```

See Also

[Recipe 1.6](#)

1.13 Building Equality Functionality into Your Own Types

Problem

You have your own structs and classes and you want to build equality-checking functionality into them.

Solution

Build your equality functionality into the protocols to which your types conform. This is the way to go!

Discussion

Let me give you an example. Let's say that we have a protocol called Named:

```
protocol Named{
  var name: String {get}
}
```

We can build the equality functionality into this protocol. We can check the name property and if the name is the same on both sides, then we are equal:

```
func ==(lhs : Named, rhs: Named) -> Bool{
  return lhs.name == rhs.name
}
```

Now let's define two types, a car and a motorcycle, and make them conform to this protocol:

```
struct Car{}
struct Motorcycle{}

extension Car : Named{
  var name: String{
    return "Car"
  }
}

extension Motorcycle : Named{
  var name: String{
    return "Motorcycle"
  }
}
```

That was it, really. You can see that I didn't have to build the equality functionality into Car and into Motorcycle separately. I built it into the protocol to which both types conform. And then we can use it like so:

```
func example1(){

  let v1: Named = Car()
  let v2: Named = Motorcycle()

  if v1 == v2{
    print("They are equal")
  } else {
    print("They are not equal")
  }
}
```

This example will say that the two constants are not equal because one is a car and the other one is a motorcycle, but what if we compared two cars?

```
func example2(){

  let v1: Named = Car()
  let v2: Named = Car()
```

```

    if v1 == v2{
        print("They are equal")
    } else {
        print("They are not equal")
    }
}
}

```

Bingo. Now they are equal. So instead of building the equality functionality into your types, build them into the protocols that your types conform to and you are good to go.

See Also

[Recipe 1.12](#)

1.14 Looping Conditionally Through a Collection

Problem

You want to go through the objects inside a collection conditionally and state your conditions right inside the loop's statement.

Solution

Use the new `for x in y where` syntax, specifying a `where` clause right in your `for` loop. For instance, here I will go through all the keys and values inside a dictionary and only get the values that are integers:

```

let dic = [
    "name" : "Foo",
    "lastName" : "Bar",
    "age" : 30,
    "sex" : 1,
]

for (k, v) in dic where v is Int{
    print("The key \(k) contains an integer value of \(v)")
}

```

Discussion

Prior to Swift 2.0, you'd have to create your conditions *before* you got to the loop statement—or even worse, if that wasn't possible and your conditions depended on the items inside the array, you'd have to write the conditions *inside* the loop. Well, no more.

Here is another example. Let's say that you want to find all the numbers that are divisible by 8, inside the range of 0 to 1000, inclusively:

```
let nums = 0..<1000
let divisibleBy8 = {$0 % 8 == 0}
for n in nums where divisibleBy8(n){
    print("\(n) is divisible by 8")
}
```

And of course you can have multiple conditions for a single loop:

```
let dic = [
    "name" : "Foo",
    "lastName" : "Bar",
    "age" : 30,
    "sex" : 1,
]

for (k, v) in dic where v is Int && v as! Int > 10{
    print("The key \(k) contains the value of \(v) that is larger than 10")
}
```

See Also

[Recipe 1.11](#)

1.15 Designing Interactive Interface Objects in Playgrounds

Problem

You want to design a view the way you want, but don't want to compile your app every time you make a change.

Solution

Use storyboards while designing your UI, and after you are done, put your code inside an actual class. In IB, you can detach a view so that it is always visible in your playground while you are working on it, and any changes you make will immediately be shown.

Discussion

Create a single-view app and add a new playground to your project, as shown in [Figure 1-6](#).

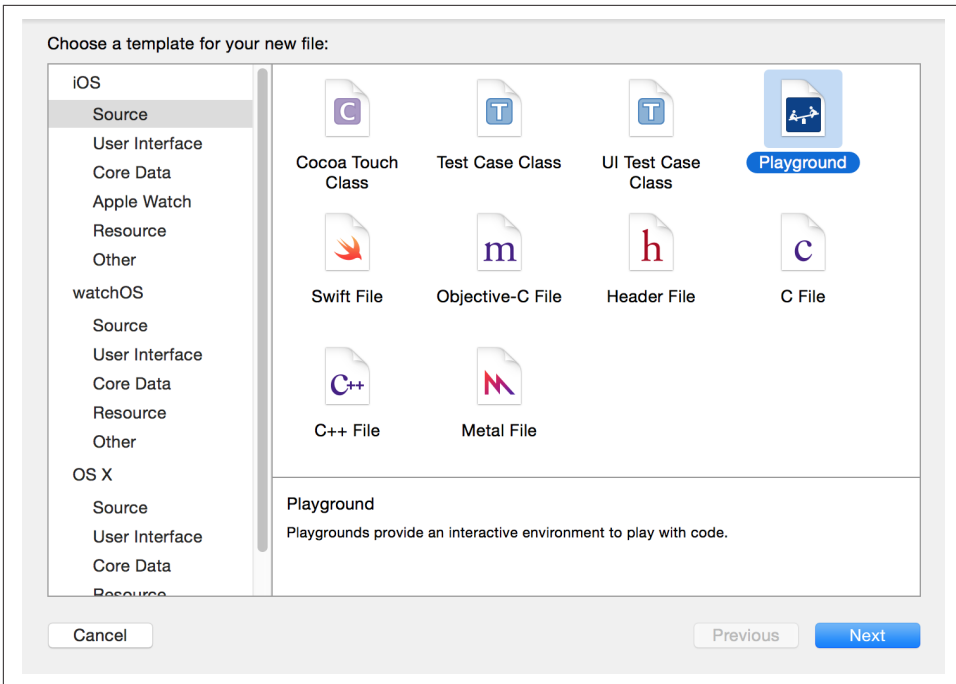


Figure 1-6. Add a new playground to your project

Write code similar to this to create your view:

```
import UIKit

var view = UIView(frame: CGRect(x: 0, y: 0, width: 300, height: 300))
view.backgroundColor = UIColor.greenColor()
```

Now on the right hand side of the last line of code that you wrote, you should see a + button. Click that (see Figure 1-7).



Figure 1-7. Click the little + button to get your view right onto your playground

By clicking that button, you will get a live preview of your view inside your playground. Now you can continue changing your view's properties and once you are done, add a new preview of your view, so that you can compare the previous and the new states (see Figure 1-8). The first view shown has only the properties you assigned

to it up to the point that view was drawn. The second view has more properties, such as the border width and color, even though it is the same view instance in memory. However, because it is drawn at a different time inside IB, it shows different results. This helps you compare how your views look before and after modifications.



Figure 1-8. Two versions of a view

See Also

[Recipe 1.7](#)

1.16 Grouping Switch Statement Cases Together

Problem

You want to design your cases in a `switch` statement so that some of them fall through to the others.

Solution

Use the `fallthrough` syntax. Here is an example:

```
let age = 30

switch age{
case 1...10:
    fallthrough
case 20...30:
    print("Either 1 to 10 or 20 to 30")
default:
    print(age)
}
```



This is just an example. There are better ways of writing this code than to use `fallthrough`. You can indeed batch these two cases together into one case statement.

Discussion

In Swift, if you want one case statement to fall through to the next, you have to explicitly state the `fallthrough` command. This is more for the programmers to look at than the compiler, because in many languages the compiler is able to fall through to the next case statement if you just leave out the `break` statement. However, this is a bit tricky because the developer might have just forgotten to place the `break` statement at the end of the case and all of a sudden her app will start behaving really strangely. Swift now makes you request fall-through explicitly, which is safer.

1.17 Bundling and Reading Data in Your Apps

Problem

You want to bundle device-specific data into your app. At runtime, you want to easily load the relevant device's data and use it without having to manually distinguish between devices at runtime.

Solution

Follow these steps:

1. In your asset catalogue, tap the + button and create a new Data Set (see [Figure 1-9](#)).

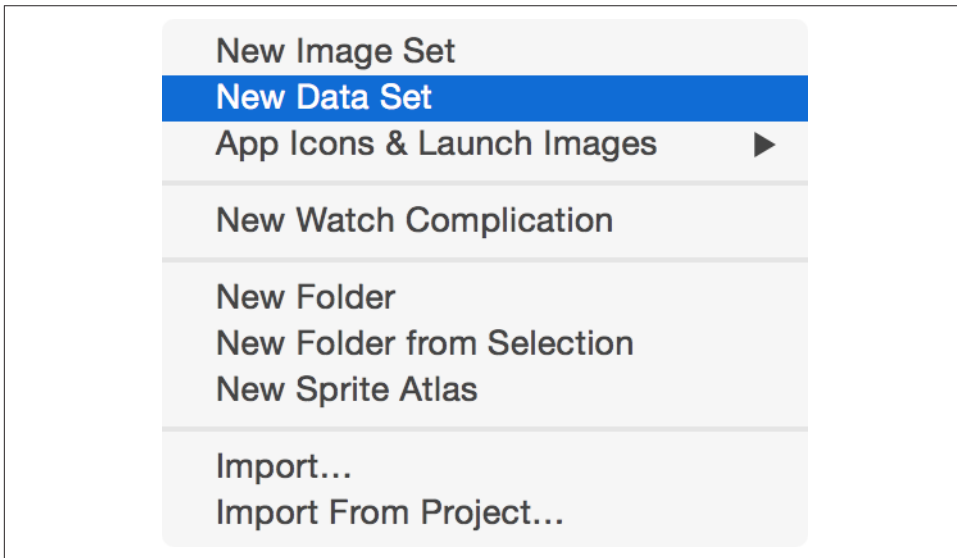


Figure 1-9. Data sets contain our raw device-specific data

2. In the Attributes inspector of your data set, specify for which devices you want to provide data (see [Figure 1-10](#)).

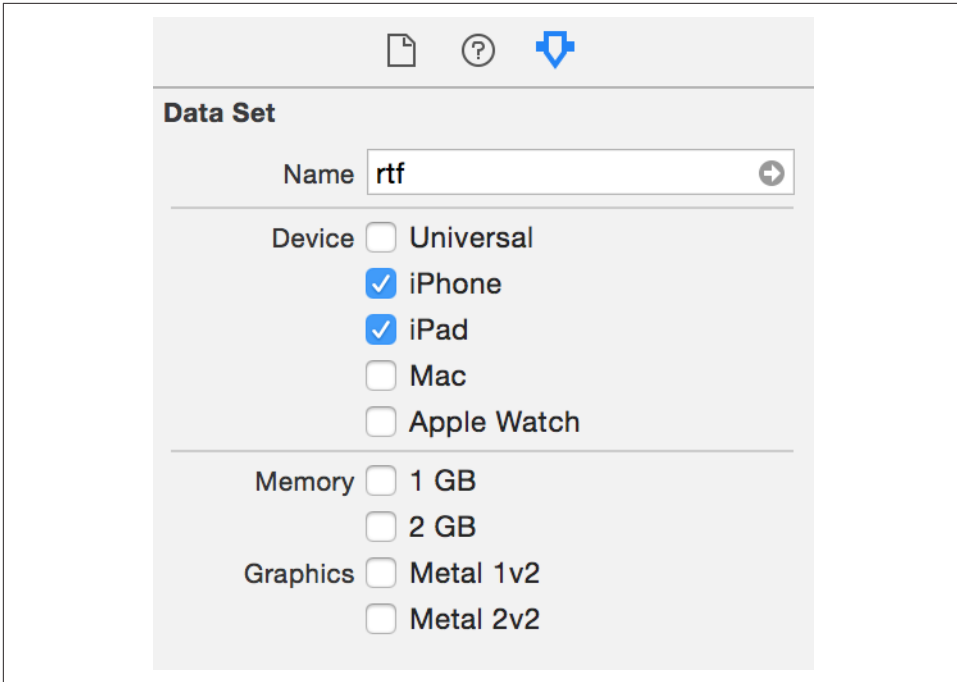


Figure 1-10. I have chosen to provide data for the iPad and iPhone in this example

3. Drag and drop your actual raw data file into place in IB
4. In your asset list, rename your asset to something that you wish to refer it to by later (see [Figure 1-11](#)).

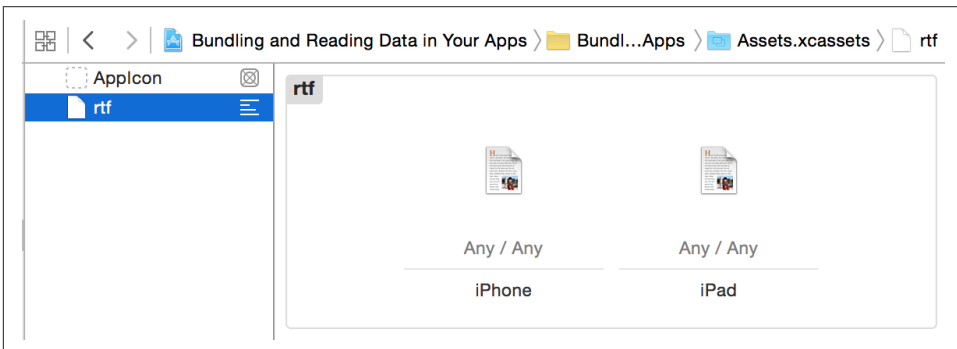


Figure 1-11. I have placed two RTF files into this data asset: one for iPhone and another for iPad



In the iPhone RTF I've written "iPhone Says Hello," and the iPad one says "iPad Says Hello"; the words iPhone and iPad are bold (attributed texts). I am then going to load these as attributed strings and show them on the user interface (see [Figure 1-13](#)).

5. In your code, load the asset with the `NSDataAsset` class's initializer.
6. Once done, use the `data` property of your asset to access the data.

Discussion

Place a label on your UI and hook it up to your code under the name `lbl` (see [Figure 1-12](#)).

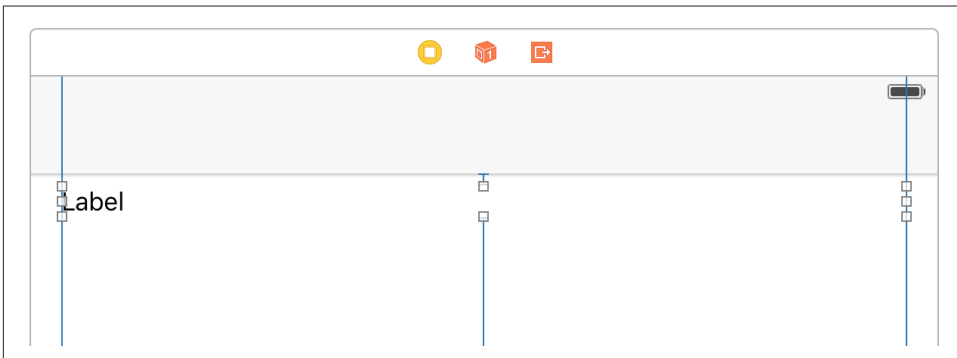


Figure 1-12. Place a label on your user interface and add all the constraints to it (Xcode can do this for you). Hook it up to your code as well.

Then create an intermediate property that can set your label's text for you:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var lbl: UILabel!

    var status = ""{
        didSet{lbl.text = status}
    }

    ...
}
```

When the view is loaded, attempt to load the custom data set:

```
guard let asset = NSDataAsset(name: "rtf") else {
    status = "Could not find the data"
    return
}
```



The name of the data asset is specified in the asset catalogue (see [Figure 1-11](#)).

Because data assets can be of any type (raw data, game levels, etc.), when loading an attributed string, we need to specify what type of data we are loading in. We do that using an *options* dictionary that we pass to NSAttributedString's constructor. The important key in this dictionary is `NSDocumentTypeDocumentAttribute`, whose value in this case should be `NSRTFTextDocumentType`. We can also specify the encoding of our data with the `NSCharacterEncodingDocumentAttribute` key:

```
let options = [  
    NSDocumentTypeDocumentAttribute : NSRTFTextDocumentType,  
    NSCharacterEncodingDocumentAttribute : NSUTF8StringEncoding  
] as [String : AnyObject]
```

Last but not least, load the data into our string and show it (see [Figure 1-13](#)):

```
do{  
    let str = try NSAttributedString(data: asset.data, options: options,  
        documentAttributes: nil)  
    lbl.attributedText = str  
} catch let err{  
    status = "Error = \(err)"  
}
```

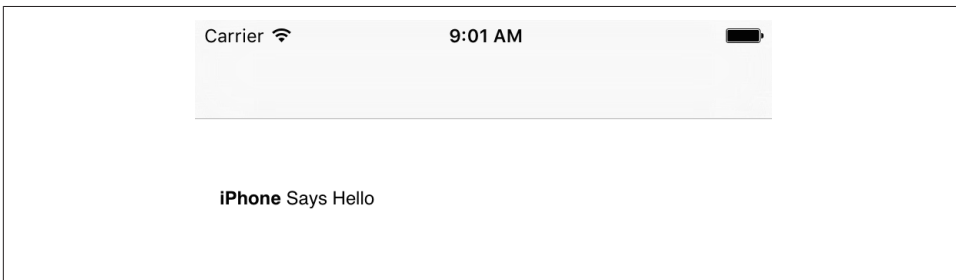


Figure 1-13. This is how my string looked when I saved it in RTF format and now it is loaded into the user interface of my app

See Also

[Recipe 1.6](#)